Solaris System Call Emulation in a Complete Machine Simulator

Bill Clarke,
(with Peter Strazdins, Andrew Over, Adam Czezowski)

Department of Computer Science,
Australian National University,
and
The CC-NUMA Project (ANU/Sun/Gaussian)


13 August 2003
1 Talk Outline

- **Sparc-Sulima**: complete machine simulator for SPARC V9
- **Why System Call Emulation?**
- **How traps and system calls work**
- **Solemn**: Solaris System Call Emulation
  - structure of Solemn
  - the Solemn nucleus
  - traps and system calls
  - system call emulation
  - reentrant system call handlers
  - example of reentrant system call: read
  - files
  - memory management
  - Solaris types, structures, values and conversions
  - executable loading and dynamically linked executables
  - current status and benchmarks, bugs and todo
2 Sparc Sulima

- SPARC-V9 ISA complete machine simulator
- currently targeting UltraSPARC I and II (plans for USIII)
- Written in C++, with some SPARC assembler for optimisations
- Python scripting interface (via SWIG)
- Efficient instruction decoding via a SLED specification of SPARC V9
- Instruction decode is cached (as part of simulated I$)
- Other optimisations: GPR caching, SPARC-on-SPARC simulation
- Portable to big-endian platforms (tested on PowerPC), plans to port to little-endian platforms
3 Why System Call Emulation?

- Already have OS-Emulation mode (UserSim):
  - has its own C-library (from RSIM, based on glibc)
  - quite limited in what system calls are provided (no mmap!)
  - results not identical to host executables (e.g., floating point printf)
- Complete machine boot so far unsuccessful:
  - not enough experience in low level boot sequence
  - we are still working on it, new student is keen and experienced
- Gain experience in internals of Solaris: useful for future full machine boot
- Implementing our own memory manager was an interesting challenge
- Easier to change architecture (e.g., MMU, cache sizes) than with complete machine simulation (would likely require OS modifications)
4 Traps and System calls

- Solaris user programs communicate with the operating system call via system calls; these are implemented using the $Tcc$ instruction.
- A $Tcc$ instruction contains a trap number (0 to 127).
- Solaris defines operating system actions given different trap numbers:
  - SunOS 4.x, Solaris 32-bit and Solaris 64-bit system calls
    - $%g1$ determines the actual system call
    - $%o0 \ldots %o5$ are the parameters
    - $%o0 (+$ possibly $%o1)$ are the return values
    - return value is an error number if the condition code register is set
    - flush/clean register windows, 32-bit get/set CCR, fast system calls (e.g., gethrtime, gettimeofday)
- Solaris defines 231 system calls (32 and 64-bit)
5 Structure of Sparc-Sulima with Solemn
6 The Solemn Nucleus

- written completely in SPARC assembler
- resides in ROM of the simulated machine at RSTVaddr
- power-on reset (boot) gets handled by the nucleus:
  - initialise standard trap base address (TBA)
  - initialise MMU and caches
  - call out to Solemn, asking it to load the executable
  - set up user state (program counter = entry point, stack pointer, ...)
  - jump to user-level entry point using **RETRY**
- standard trap handlers also exist, at the TBA:
  - register window management
  - MMU data and instruction misses, data protection faults
7 System call emulation

- **Solemn** intercepts Tcc instructions and emulates the effect
- Solemn uses 3 trap numbers to communicate with the nucleus
- Solemn currently implements 24 of the 231 Solaris system calls. e.g.,
  - process related: exit, getpid, getuid, getgid
  - file related: read, write, open, close, stat, lseek
  - memory related: brk, mmap, munmap, memcntl
- Some system calls are trivial to implement: getpid, close, lseek
- Many are a little harder, requiring re-entrant system call handlers: read, write, open, stat
- Some are much harder, requiring additional infrastructure: threads, signals
- Some do not make sense to implement for Solemn: fork
8 Reentrant system call handlers

- Solemn communicates with the simulation’s memory via the MMU
- This can cause exceptions: data page miss or protection fault
- This exception must be handled before the system call can continue:
  - generate the exception and return control to the simulator
  - exception handled by the nucleus (calls a special Solemn system call)
  - the TTE is entered into the TLB and the faulting instruction is retried
- In general, the system call should restart where it failed
- Host buffer and state stored in a ReentryBuffer until system call complete
- Future threading: multiple ReentryBuffers, indexed by CPU id
9 Reentrant system calls: read

**system call**: read(int fd, void *buf, size_t nbyte):

Solemn::SysCallRet Solemn::handle_sys_read(SPARCV9ExternalHelper& ext) {
    Files::FD fd = ext.get_o0();
    VA buf = ext.get_o1();
    UInt64 nbyte = ext.get_o2();

    Ptr<ReentryBuffer>& reentry_buffer =
        lookup_reentry_buffers(ext, fd, buf, nbyte, 0, buf, nbyte);
    if (reentry_buffer->get_phase() == 0) {
        Int64 nread = files->read(fd, reentry_buffer->get_buf(), nbyte);
        if (nread == files->err) {
            reentry_buffer.reset();
            return sys_error(files->errno_);
        }
        reentry_buffer->set_length(nread);
        reentry_buffer->set_phase(1);
    }
    int exc = reentry_buffer->memcpy_to_sim(ext);
    if (exc) return sys_exception(exc);
    Int64 nread = reentry_buffer->get_length();
    reentry_buffer.reset();
    return sys_ok(nread);
}
10 Files wrapping

- **Files**: wrapper around host IO files, file descriptors
- Includes its own file descriptors (mapped to host file descriptors)
- Implements all standard IO calls: `read, write, ...`
- Contains an error number (copy of host error number, updated when necessary)
- Plan to integrate “change root” code from visiting student
11 Memory management

- **MemoryManager**: responsible for all handling memory-related
  - **traps**: page misses and protection faults; and
  - **system calls**: brk, mmap, munmap, ...

- Contains **FreeSpace and VAMappings managers**:
  - **FreeSpace**:
    - unused virtual address intervals
    - free space search (for unfixed mmap)
  - **VAMappings**:
    - current virtual address mappings and host pointers
    - physical address assignments: if (virtual) page is on RAM
    - reverse lookup: given RAM page (PA), get virtual address
    - page replacement: virtual memory (least recently used)

- Currently, pages are fixed at 8KB, but I have plans for multiple page sizes (as per Solaris 9)
Solaris types and values and conversions

- Aim is for Solemn to be runnable on other platforms (as host)
- Need to convert to and from host, Solaris 32 or 64-bit, and intermediate (Solemn internal)
  - types: size_t, offset_t, id_t, ...
  - structures: struct stat, struct stat64 ...
  - values: O_RDONLY ..., SEEK_SET ..., MAP_SHARED ...
- Solaris types, structures and values are declared within 3 structures:
  - Sol: types, structures and values common to both 32 and 64-bit Solaris
  - Sol32, Sol64: inherit from Sol, architecture specific types and structures
- SolConvert: a namespace containing functions for converting to and from simulated types and values and host types and values
- Intermediate structures: OpenFlags, MMapFlags, Prots, ...
13 File loading and dynamically linked executables

- **ElfLoader**: has the job of loading a file
- Uses the **MemoryManager** to mmap parts of the file
- Has to deal with:
  - determines whether the executable is 32 or 64-bit
  - interpreter: the executable is dynamically linked
  - program header table entry (only if dynamically linked)
  - if it is a dynamic/shared object (does not have a base address):
    - need to determine range of addresses used; and
    - determine a virtual address offset to load it at
  - zero-filling the last parts of each segment
- **Solemn** uses at least one, and at most two ElfLoaders
Initialisation

Called by the nucleus:
- Create files wrapper
- Determine executable arch
- Initialise memory manager (requires arch)
- Load executable (requires memory manager)
- If it is dynamically linked (has interpreter)
  - set up various auxiliary vector entries (e.g., entry point)
  - load interpreter (check it has same arch)
  - set up other auxiliary vector entries (e.g., base address of interpreter)
- Create the stack (requires auxiliary entries)
- Set stack pointer and entry point for nucleus
15 Benchmarks

- empty.c: (i.e., int main() { return 0; })
- UserSim linked: 258 instructions (mostly nucleus boot)
- Solemn statically linked: 649 instructions (about half nucleus)
- Solemn dynamically linked: 310593 instructions
- [FIXME: add more benchmarks: EMXIF]
16 Bugs, problems, and todo

- **Bugs:**
  - 64-bit dynamically linked executables die partway through linking

- **Problems:**
  - Debugging: Sparc-Sulima is large: over 40000 lines of code, of which 6000 are Solemn

- **Todo:**
  - more system calls!
  - threads!
  - Gaussian kernels!